Université Bordeaux 1 Licence S&TS Mention Informatique S5 2011/2012

UE INF5011 Programmation 3

Programmation Fonctionnelle et Symbolique

Devoir surveillé No 2

Tous documents autorisés. Mercredi 23 Novembre 2011 Durée : 1h20.
Le barème est donné à titre **indicatif**.

Exercice 1 *(8pts)*

On dispose d'un vecteur (tableau à une dimension) de taille arbitraire tel que la case `i` contient la chaîne de caractères correspondant à l'écriture du nombre `i` en toutes lettres. À partir de ce vecteur, on souhaite construire une table de hachage permettant de traduire un nombre entier écrit en toute lettres (chaîne de caractères) en sa valeur numérique.

1. Écrire une fonction `make-number-table (vector)` qui retourne une telle table. Exemples :

```
CL-USER> *vecteur-nombres*
#("zero" "un" "deux" "trois" "quatre" "cinq" "six" "sept" "huit" "neuf")
CL-USER> (setf *table-nombres*  (make-number-table *vecteur-nombres*))
#<HASH-TABLE :TEST EQUAL :COUNT 10 {1003367E71}>
CL-USER> (gethash "trois" *table-nombres*)
3
T
CL-USER> (gethash "cent" *table-nombres*)
NIL
NIL
```

On dispose d'une table de hachage traduisant des nombres entiers écrits en toutes lettres en leur valeur numérique. On suppose que si la table contient n nombres, elle contient les nombres de 0 à $n-1$.

2. Écrire une fonction `vector-from-table (table)` qui à partir d'une telle table reconstruit le vecteur d'origine. Exemple :

```
CL-USER> (vector-from-table *table-nombres*)
#("zero" "un" "deux" "trois" "quatre" "cinq" "six" "sept" "huit" "neuf")
```

Exercice 2 *(4pts)*

Écrire une fonction **itérative** `to-vector (l1 l2)` qui étant données deux listes de longueurs quelconques, construit un vecteur de dimension le minimun des longueurs des deux listes et tel que le $i$ème élément du vecteur est une liste constituée du $i$ème élément de `l1` et du $i$ème élément de `l2`. Exemples :

```
CL-USER> (to-vector '() '())
#()
CL-USER> (to-vector '(1 2 3 4) '(a b c))
#((1 A) (2 B) (3 C))
CL-USER> (to-vector '(1 2 3) '(a b c d e))
#((1 A) (2 B) (3 C))
```

Exercice 3 *(8pts)*

On souhaite manipuler des expressions arithmétiques en notation préfixe. Une *expression arithmétique* est

- soit une *variable* (représentée par un symbole `Lisp`),
- soit un *nombre* (représenté par un nombre),
- soit une *expression composée* représentée par une liste `(op ex1 ex2 ... exn)` telle que `op` est un symbole d'opération et chaque `exi`, une expression.

Les opérations possibles sont l'addition et la multiplication (n-aires, associatives et commutatives), l'exponentiation (binaire) représentées respectivement par les symboles `+, *` et `exp`.

Exemples d'expressions :

```
2    (+ x y)      (+ x 3 (* y 4) 2 z (+ x 1))
x    (exp x 2)    (+ 2 (exp x 3) (* x y z))
```

Quand les arguments d'une opération associative et commutative sont tous des variables, on souhaite les regrouper en fonction du type de l'opérateur :

```
(+ x y z x z y x z)  →  (+ (* Y 2) (* X 3) (* Z 3))
(* x y z x z y x z)  →  (* (EXP Y 2) (EXP X 3) (EXP Z 3))
```

1. Écrire une fonction `compress-vars (exp op)` qui permette de réaliser ce type de regroupements. On pourra utiliser la fonction `count` dont la documentation `HyperSpec` est donnée Fig. **??** en annexe page **??**.

Exemples :

```
CL-USER> (compress-vars '(+ x y z x z y x z) '*)
(+ (* Y 2) (* X 3) (* Z 3))
CL-USER> (compress-vars '(* x y z x z y x z) 'exp)
(* (EXP Y 2) (EXP X 3) (EXP Z 3))
```

2. Écrire une fonction `partition-args (args)` qui prend en paramètre la liste des arguments d'une expression arithmétique et retourne trois valeurs :
   (a) la liste des arguments qui sont des nombres
   (b) la liste des arguments qui sont des variables
   (c) la liste des arguments qui sont des expressions composées

Exemples :

```
CL-USER> (partition-args '(x 3 (* y 4) 2 z (+ x 1)))
(3 2)
(X Z)
((* Y 4) (+ X 1))
CL-USER> (partition-args '(2 (exp x 3) (* x y z)))
(2)
NIL
((EXP X 3) (* X Y Z))
```

FIN

```
Function COUNT, COUNT-IF, COUNT-IF-NOT

Syntax:
count item sequence &key from-end start end key test test-not => n
count-if predicate sequence &key from-end start end key => n
count-if-not predicate sequence &key from-end start end key => n

Arguments and Values:

item---an object.

sequence---a proper sequence.

predicate---a designator for a function of one argument that returns
            a generalized boolean.
from-end---a generalized boolean. The default is false.

test---a designator for a function of two arguments that returns
       a generalized boolean.
test-not---a designator for a function of two arguments that returns
           a generalized boolean.
start, end---bounding index designators of sequence.
             The defaults for start and end are 0 and nil, respectively.
key---a designator for a function of one argument, or nil.

n---a non-negative integer less than or equal to the length of sequence.

Description:

count, count-if, and count-if-not count and return the number of elements
in the sequence bounded by start and end that satisfy the test.

The from-end has no direct effect on the result.
However, if from-end is true, the elements of sequence will be supplied
as arguments to the test, test-not, and key in reverse order, which may
change the side-effects, if any, of those functions.

Examples:
 (count #\a "how many A's are there in here?") =>  2
 (count-if-not #'oddp '((1) (2) (3) (4)) :key #'car) =>  2
 (count-if #'upper-case-p "The Crying of Lot 49" :start 4) =>  2
```

FIGURE 1 – Documentation HyperSpec de la fonction count